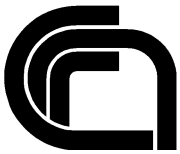

Consiglio Nazionale delle Ricerche

Enhancing RDAP searching and filtering capabilities

M. Loffredo, M. Martinelli

IIT TR-07/2018

Technical Report

Ottobre 2018

Istituto di Informatica e Telematica

Summary

Abstract

In addition to sorting and paging, the possibility to submit powerful search queries is considered one of the most effective means to improve the management of REST APIs outputs. Searching and filtering are two of the REST implementation principles addressing the problem of retrieving a set of results matching certain criteria. At present, RDAP provides very limited search capabilities, which make the extraction of the desired information a time consuming process. In order to fix such inefficiency, this document defines two new parameters, which enable the user to submit complex queries and filters.

1. Introduction

In addition to sorting and paging, the capability to submit powerful search queries is considered one of the most effective means to improve the management of REST APIs results ([REST]). The main reasons for its presence are:

- improving the precision of the queries and, consequently, obtaining more reliable results;
- speeding up the interaction between client and server;
- minimizing the bandwidth usage;
- decreasing CPU time and memory spent on both server and client.

At present, RDAP ([RFC7482]) provides very limited search capabilities because a search query consists of a single pattern. Therefore, a search query can potentially generate a large result set that, in the best case scenario, must be scrolled with the aim to look for the desired data but, in the worst case scenario, can be truncated, according to the server limits.

Even if a server provided clients with operators for result sorting and paging, the extraction of the desired information from a result set could be very time consuming. Furthermore, users might be interested in performing searches that currently RDAP does not allow. For example, a user might search all the domains whose names match a specific pattern and registration dates fall within a time period.

In authors' opinion, the best solution to fix such inefficiency is to enhance searching and filtering capabilities at server side. In this manner, clients could obtain always and only the desired results with the minimum response. This specification extends RDAP with new query parameters to improve its searchability.

2. Searching vs. filtering

Both searching and filtering seem to address the same problem of retrieving a set of results matching certain criteria, but, in RESTful context, they have two distinct meaning:

- *searching* is done as a first step of a retrieval process. It consists of a query containing data attributes, which are usually mapped onto the underlying database fields, which have a high degree of uniqueness and whose values are scanned through an index (e.g. unique keys);
- *filtering* is applied on top of a search to narrow down the result set. It consists of a set of additional conditions on those data attributes, which are usually mapped on the underlying database fields whose values are scanned sequentially (e.g. dates or enum type columns);

The difference above derives from the following facts:

- Tipically, the process of finding relevant data consists of a sequence of refining steps. The user can start with a rather general search and then applies additional conditions to the initial criterion in order to get an increasingly smaller result set until to obtain the relevant data.

- It is well known to the implementers that the efficiency of a retrieval process is affected by both the kind and the evaluation order of the search conditions. The constraints put on indexed data attributes should be evaluated first in order to return quickly a relatively small result set, which can be scanned sequentially and/or furtherly restricted. On the contrary, such fact is unknown to users who basically make no difference about the search conditions and rely on applications capabilities to find the relevant data as fast as possible.

- As a consequence, data retrieval applications should provide users with a high degree of searchability but, at the same time, should mitigate the risks to submit an inefficient query to the underlying DBMS (i.e. a query taking too longer or requiring too many resources).

- If we divide the set of the searchable data attributes in two groups, those enabling a “fast retrieval” and those causing a “slow retrieval”, it would be recommendable to implement two different operators (i.e. the searching operator and the filtering operator) each allowing users to create search conditions based on the attributes of one group. The condition based on the first group attributes is required while the condition based on the second group attributes is optional. This solution reduces the risk to produce inefficient queries to the underlying DBMS.

Obviously, the mere presence of a searching operator does not ensure that the query generated and sent to the DBMS will be efficient, if partial matching is allowed. In this case, the application is required to control the use of partial matching (e.g. by avoiding wildcard prefixed patterns).

3. REST query languages

The availability of a REST service capability to submit complex queries involves the definition of a REST query language. Some REST query languages are available on the web:

- RSQL ([RSQL]) is a query language for parametrized filtering of entries in RESTful APIs. RSQL is simple and compact. Anyway, in the authors’ opinion, it would be preferable a query syntax based on widely shared data format like JSON ([RFC7159]). In addition, RSQL expressions can contain only primitive values; objects and arrays are not considered.

- RESTDB ([RESTDB]) is an example of REST query language based on MongoDB JSON query syntax. This query language does consider only primitive values too. In addition, the syntax to represent a basic search predicate (i.e. property-operator-value triple) does not seem to be intuitive.

- GraphQL ([GRAPHQL]) is a query language, specification, and collection of tools, designed to operate over a single endpoint via HTTP. GraphQL allows the user to implement highly query-able REST APIs, which serve requests asking for small and different data. Anyway, this does not seem to be the RDAP scenario because users tend to ask for the same set of strongly related data attributes as described in

[I-D.loffredo-regext-rdap-partial-response]. In addition, an approach based on the adoption of GraphQL would not be compliant with the current RDAP status.

In the authors' opinion, the query language should be based on JSON format, should take into account all data types and should be both human and machine processable.

4. RDAP Path Segment Specification

According to the principles stating the distinction between searching and filtering, we define two new RDAP query parameters:

- ***query***: it allows the user to submit a complex search. It must be used in place of a RDAP search path segment (e.g. domains?name). Therefore, the query strings "domains?name=….." and "domains?query=….." would be both valid while the query string "domains?name=…..&query=….." would not.

- ***filter***: it allows the user to filter the results according to the values of those RDAP properties that are not used as search path segments (e.g. status or roles). It can be used in addition to either a search path segment or a query parameter. Therefore, both the query strings "domains?name=…..&filter=….." and "domains?query=…..&filter=……" would be valid while the query string "domains?filter=….." would not.

In both cases, the parameter value is a JSON ([RFC7159]) expression, which enable clients to submit search conditions whose complexity ranges from very simple to extremely complicated.

Traditionally, a search condition includes a set of basic predicates joined by the logical operators "and", "or", and "not". A predicate contains three components:

- a property name;
- an allowed operator for the property;
- a filter value whose type is allowed for the property.

In case of query parameter, the property name corresponds to a search path segment as defined in RFC7482 ([RFC7482]):

- Domains: name, nsLdhname , nsIp
- Nameservers: name, ip
- Entities: fn, handle

Other search path segments can be considered as reported in [I-D.loffredo-regext-rdap-reverse-search].

As for filter, in [I-D.loffredo-regext-rdap-sorting-and-paging], the authors have already defined a set of RDAP property references ([RFC7483]) involved in the specification of sort criteria that can be adopted in the specification of a filter as well. In addition to those properties, the multivalued properties, such as roles and status, have been added. The properties are:

- Object common properties:
    - registrationDate
    - reregistrationDate
    - lastChangedDate

    - expirationDate
    - deletionDate
    - reinstantiationDate
    - transferDate
    - lockedDate
    - unlockedDate

- Object specific properties:
    - Domain: status
    - Entity: org, email, voice, country, cc, city, roles, status

In addition to the logical operators, it is reasonable to expect the presence of commonly used comparison operators like “equal”, “not equal”, “less than” and so on. Specific operators on strings like “contains”, “starts with”, “ends with” can be implemented using the “equal” operator and the wildcard character. Appropriate operators on arrays should be considered, too. In the following list, the operators are shown:

- no values: isnull, isnotnull
- one value: eq, ne, le, ge, lt, gt
- array of two values: between
- array of N values: in, notin

Finally, value types can be “string”, “number”, “boolean”, “datetime”, “array” (of primitive type) or “object”. An object value occurs when the property can be described in terms of sub-properties (e.g. entityAddr). In this case, the constraints on the sub-properties are implicitly joined by "and". The only operators available for an object value are “eq” and “ne”. As regards to datetime values, RFC3339 full-date and date-time formats should be supported ([RFC3339]).

As a consequence of what stated above, the simplest JSON expression describing a predicate consists of an array of three items. A complex predicate can be represented through a “one member” JSON object where the logical operator is the member name and the sub-predicates (one or more) are the member values.

Even if the deserialization of a JSON array in a data structure is not a standard capability of JSON libraries, it can be accomplished by a customization requiring a few lines of code but, on the other hand, the proposed representation for a predicate is much more compact than using a JSON object.

Below, a JCR-based representation ([JCR]) modelling the value of both query and filter parameters is presented:

```
@{root} $expression = {
       (
         $or_expression   |
         $and_expression  |
         $not_expression  |
         [ $predicate + ] |
         $predicate
        )
}

$or_expression = {
  "or" : [ $expression, $expression + ]
}
```

```
$and_expression = {
  "and" : [ $expression, $expression + ]
}

$not_expression = {
  "not" : $expression
}

$predicate = [
     /^[A-Za-z]+$/,
  (
       ("isnull"|"isnotnull")                                  |
       ("eq"|"ne"), $basic_or_object_value                     |
       ("le"|"lt"|"gt"|"ge"), $not_pattern_value               |
       "between", [ $not_pattern_value, $not_pattern_value ] |
       ("in"|"notin"|"any"|"all"|"exactly"), $array_value
     )
]

$basic_or_object_value = (
                          $basic_value  |
                          $object_value
                          )

$object_value = { // : any * }

$basic_value = @{not} (
                        { // : any * } |
                        [ any * ]      |
                        null
                       )

$not_pattern_value = @{not} (
                              { // : any * } |
                              [ any * ]      |
                              null           |
                              $pattern_value
                             )

$pattern_value = /^[^\*]*\*[^\*]*$/

$array_value = [ $not_pattern_value + ]
```

The *any*, *all* and *exactly* operators are used with the following meaning:

- *any* means that the property must contain at least one of the values in the array;
- *all* means that the property must contain all the values in the array, but it could contain also additional values;
- *exactly* means that the property must contain all the values in the array and cannot contain any additional value.

If an array contains only one value, *any* and *all* have the same meaning.

In addition to the JCR constraints, some further constraints must be applied:

- all the values in an array must have the same type;
- *any*, *all* or *exactly* operators must only be associated to an RDAP property whose type is an array of primitive types (like status, roles).

Here, in the following, some examples of predicates:

```
["registrationDate","ge","2018-01-20"]
```

```
{"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
```

```
{"not":{"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}}
```

All predicates in an array of predicates are implicitly combined by "and". Therefore, the two expressions below have the same meaning:

```
{"and":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
```

```
[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]
```

In the same way as the array of predicates is a shortcut for the definition of two or more predicates joined by "and", the *in* operator is a shortcut for the definition of a list of predicates, each one referencing the same property, joined by "or". So the following expressions are equivalent:

```
{"or":[["cc","eq","it"],["cc","eq","de"],["cc","eq","fr"]]}
```

```
["cc","in",["it","de","fr"]]
```

The operator "between" is a shortcut for two predicates joined by "and" including the same property.

```
{"and":[["registrationDate","ge","2018-01-20"],["registrationDate","le","2019-01-20"]]}
```

```
["registrationDate","between",["2018-01-20","2019-01-20"]]
```

The *isnull* and *isnotnull* operator are used in those cases where the predicate would express, respectively, the absence or the presence of a property in the expected results. For these operators, the last item of the predicate array can be missing and, if present, must be ignored. For example, the following predicate allows the user to search for the domains that have never been transferred:

```
["transferDate","isnull"]
```

Finally, the RDAP basic search is equivalent to a query parameter expression including only one predicate where the property name is the name of basic search path segment, the operator is “eq” and the value is the basic search pattern. Therefore, the following query strings have the same meaning and must return the same results:

```
domains?name=exam*.com
```

```
domains?query=["name","eq","exam*.com"]
```

4.1 Examples

Here in the following some examples of REST queries that cab me submitted to .it RDAP public test server:

```
https://rdap.pubtest.nic.it/domains?name=we*.it&filter=["registrationDate","ge","2018-01-20"]
```

```
https://rdap.pubtest.nic.it/domains?name=we*.it&filter={"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
```

```
https://rdap.pubtest.nic.it/domains?name=we*.it&filter={"not":{"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}}
```

```
https://rdap.pubtest.nic.it/domains?name=wu*it&filter=["transferDate","isnull"]
```

```
https://rdap.pubtest.nic.it/domains?query=[["name","eq","test-*.it"],["nsLdhName","eq","wns1.rtr-dev.com"]]
```

```
https://rdap.pubtest.nic.it/domains?query=[["name","eq","test-*.it"],["entityAddr","eq",{"value":{"cc":"be"},"role":"registrant"}]]&filter={"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
```

5. Filtering Metadata

According to most advanced principles in REST design, collectively known as HATEOAS (Hypermedia as the Engine of Application State) ([HATEOAS]), a client entering a REST application should use the server-provided links to dynamically discover available actions and access the resources it needs. In this way, the client is not requested to have prior knowledge of the service and, consequently, to hard code the URIs of different resources. This would allow the server to make URI changes as the API evolves without breaking the clients. Definitively, a REST service should be as self-descriptive as possible.

Therefore, the implementation of the filter parameter requires servers to provide additional information in their responses about the available filters. Such information is gathered in a new data structure named "*filtering_metadata*". There is no need for an equivalent metadata section about searching because, in that case, the property names correspond to search path segments, which are reported in some RFCs.

The data structure contains the following fields:

- *currentFilter:* the value of filter parameter as specified in the query string;
- *availableFilters:* an array of objects each one describing an available filter:
    - *property:* the name that can be used by the client to request the filter;

- *jsonPath:* the JSON Path of the RDAP field corresponding to the property.

6. RDAP conformance

Servers returning the "*filtering_metadata*" section in their responses must include "*filtering_level_0*" in the rdapConformance array.

7. Implementation Considerations

The implementation of the new parameter is technically feasible, as operators for searching and filtering rows, according to a search condition are currently supported by Relational as well as NoSQL DBMSs.

Some further considerations must be made:

- RDAP does not specify any required object property. Except for *objectClassName* and the property identifying uniquely an object (e.g. *ldhName* for domain), all other properties are optional. Therefore, a client could refer to a path segment or a property that are not implemented. In those cases, the server is recommended to return an error response.

- The present document only reports some suitable filtering properties of the topmost objects in the search results array. Obviously, they can be extended with other properties that have not yet been considered.

- Servers could implicitly filter results according to user access levels. For example, registrar users could search only their own domains. The implicit filter can be represented in the same way as the explicit filter so that `<real filter>={"and":[<implicit filter>,<explicit filter>]}`

7.1 JSON in URLs

Many web services, including RDAP, rely on the HTTP GET method to take advantage of some of its features:

- GET requests can be cached;
- GET requests remain in the browser history;
- GET requests can be bookmarked.

Sometimes, it happens that such advantages should be combined with the requirement to pass objects and arrays in the query string. JSON is the best candidate as data interchange format, but it contains some characters that are forbidden from appearing in a URL. Anyway, escaping the invalid characters is not an issue because, on the client side, modern browsers automatically encode URLs and, on the server side, a lot of URL encoding/decoding libraries for all web development programming languages are available. The downside of URL encoding is that it can make a pretty long URL, which, depending on the initial length and the number of invalid characters, might exceed the practical limit of web browsers (i.e. 2,000 characters).

Other solutions to pass a JSON expression in a URL could be:

- converting JSON to Base64 ([RFC4648]), but binary data are unreadable;

- using a JSON variation that complies with URL specifications and maintains readability like Rison ([RISON]), URLON ([URLON]) or JSURL ([JSURL]).

The extensions proposed in this document rely on URL encoding because it is widely supported and the risk to exceed the maximum URL length is considered to be very unlikely in RDAP.

8. Security Considerations

A search query typically requires more server resources (such as memory, CPU cycles, and network bandwidth) when compared to lookup query. This increases the risk of server resource exhaustion and subsequent denial of service due to abuse. This risk can be mitigated by the following actions:

- limiting the rate of search requests;
- truncating the results in the response;
- providing partial responses;
- enhancing searching and filtering capabilities.

9. References

9.1 Normative References

[RFC3339] KLYNE G. AND C. NEWMAN,"DATE AND TIME ON THE INTERNET: TIMESTAMPS", DOI 10.17487/RFC3339, JULY 2002, <HTTPS://TOOLS.IETF.ORG/HTML/RFC3339>

[RFC5988] M. NOTTINGHAM,"WEB LINKING", DOI 10.17487/RFC5988, OCTOBER 2010, <HTTPS://TOOLS.IETF.ORG/HTML/RFC5988>

[RFC7159] T. BRAY, "THE JAVASCRIPT OBJECT NOTATION (JSON) DATA INTERCHANGE FORMAT", DOI 10.17487/RFC7159, MARCH 2014, <HTTPS://WWW.RFC-EDITOR.ORG/INFO/RFC7159>

[RFC7482] A. NEWTON AND S. HOLLENBECK, "REGISTRATION DATA ACCESS PROTOCOL (RDAP) QUERY FORMAT", RFC 7482, DOI 10.17487/RFC7482, MARCH 2015, <HTTP://WWW.RFC-EDITOR.ORG/INFO/RFC7482>.

[RFC7483] A. NEWTON AND S. HOLLENBECK, "JSON RESPONSES FOR THE REGISTRATION DATA ACCESS PROTOCOL (RDAP)", RFC 7483, DOI 10.17487/RFC7483, MARCH 2015, <HTTP://WWW.RFC-EDITOR.ORG/INFO/RFC7483>.

9.2 Informative References

[GRAPHQL] STURGEON P., "GRAPHQL VS REST: OVERVIEW", 2017, <HTTPS://PHILSTURGEON.UK/API/2017/01/24/GRAPHQL-VS-REST-OVERVIEW/>.

[HATEOAS] JEDRZEJEWSKI B., "HATEOAS - A SIMPLE EXPLANATION", 2018, <HTTPS://WWW.E4DEVELOPER.COM/2018/02/16/HATEOAS-SIMPLE-EXPLANATION/>.

[I-D.loffredo-regext-rdap-partial-response]
Loffredo M. and M. Martinelli, "Registration Data Access Protocol (RDAP) Partial Response", draft-loffredo-regext-rdap-partial-response-02 (work in progress), September 2018.

[I-D.loffredo-regext-rdap-reverse-search]
Loffredo M., Martinelli M. and S. Hollenbeck, "Registration Data Access Protocol (RDAP) Reverse Search Capabilities", draft-loffredo-regext-rdap-reverse-search-02 (work in progress), September 2018.

[I-D.loffredo-regext-rdap-sorting-and-paging]
Loffredo M. and M. Martinelli, "Registration Data Access Protocol (RDAP) Query Parameters for Result Sorting and Paging", draft-loffredo-regext-rdap-sorting-and-paging-05 (work in progress), September 2018.

[JCR] "JCR – JSON Content Rules", <http://json-content-rules.org/>

[JSURL] "JSURL", 2016, <https://github.com/Sage/jsurl>

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[REST] T. Fredrich, "RESTful Service Best Practices, Recommendations for Creating Web Services", <http://www.restapitutorial.com/media/RESTful_Best_Practices-v1_1.pdf>

[RESTDB] "REST API examples and queries", <https://restdb.io/docs/querying-with-the-api>

[RSQL] M. Aboullaite, "Smarter search with RSQL", <https://aboullaite.me/rsql/>, May 2018

[RISON] "Rison - Compact Data in URIs", 2017, <https://github.com/Nanonid/rison>

[URLON] "URL Object Notation", 2017, <https://github.com/cerebral/urlon>